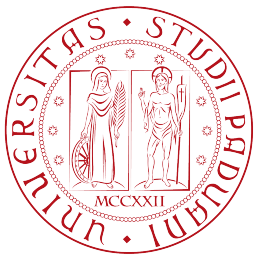

Università degli Studi di Padova

Dipartimento di Scienze Statistiche

Corso di Laurea Triennale in
Statistica per le Tecnologie e le Scienze

Ad-hoc Biomedical Information Retrieval for Global Pandemics: A Study of Methods Based on the TREC-COVID test collection

Relatore:
Prof. Massimo Melucci

Laureando:
Giacomo Virginio
1147922

Anno Accademico 2021/2022

Abstract

The TREC_COVID Challenge has the goal to create search engines to effectively and efficiently retrieve information produced at a rate never seen before, in the biomedical field.

This work focuses on the effectiveness of the information retrieval.

The search engine is based on Elasticsearch. A multitude of information retrieval techniques are tested, with the goal of identifying the ones leading to a performance improvement. The techniques' effectiveness is measured using the evaluation measures: P@20, MAP, and BPref.

The techniques explored that yield improvement in the search are: custom analyzers, filters, relevance feedback and reciprocal rank fusion. Other tested techniques, that yield negligible results, are: field boosting, bigrams and distance feature.

Ultimately, the results are compared to the ones obtained by others in the Challenge.

Contents

Chapter 1

Introduction

The outbreak of the Covid-19 global pandemic has been one of the major events of the 21th Century. Joint with the vast access to information of the last decade, it presented the need to access reliable information in a rapidly changing environment, in which new relevant information is produced and published at a rate faster than usual in the biomedical field.

Because of it, the "TREC-COVID Challenge" was created as a collaboration among the Allen Institute for Artificial Intelligence (AI2), the National Institute of Standards and Technology (NIST), the National Library of Medicine (NLM), Oregon Health & Science University (OHSU), and the University of Texas Health Science Center at Houston (UTHealth).

The challenge consists of a set of Information Retrieval test collections based on the CORD-19 dataset. It is a freely accessible and updated research dataset created in response to the White House's request to aggregate the largest structured dataset of coronavirus research for the global research community.

The challenge has two goals:

- Evaluate search algorithms and systems in order to help decision makers, belonging to medical and political fields, manage the rapidly growing corpus of COVID-19 related scientific literature;
- Discover methods to manage scientific information in future global biomedical crisis.

1.1 Challenge Structure

The TREC-COVID challenge was structured as a series of five rounds.
For each, the organizers designed a subset of the most recent CORD-19 dataset to be used in the round and released a set of *topics*, i. e., queries used for the document retrieval.

The participant would submit their runs and, after the submission deadline, NIST would use the submitted runs to produce a set of documents for each topic, so they could be assessed for relevance to the topic by biomedical field experts.

Since the CORD-19 dataset was too large to get complete judgements, the collection had to be sampled. For each topic, a sample had to be constructed in order to judge only a small fraction of the entire document set, while also identifying most of the relevant documents.

The sample sets were chosen based on the ranks that the documents had in the submitted runs for a round.

The relevance judgements produced are used to compute a run's effectiveness score.

New topics were added to the topic set in each round, which contained the cumulative set of topics. Relevance judgements of earlier rounds are free to use in the construction of runs.

To fairly evaluate the runs, they were scored against a reduced collection, removing from the most recent CORD-19 dataset all previously judged documents (the documents judged as non-relevant are also removed).

Chapter 2

Models

2.1 BM25 and BM25F

Okapi BM25 is a ranking function used by search engines to estimate the relevance of documents to a given search query.

Given a query Q containing the keywords $q_1, ..., q_n$, BM25's score of a document D is defined as follows:

$$score(D,Q) = \sum_{i=1}^{n} IDF(q_i) \cdot \frac{f(q_i,D) \cdot (k_1+1)}{f(q_i,D) + k_1 \cdot \left(1 - b + b \cdot \frac{|D|}{avgdl}\right)}$$

where $f(q_i, D)$ is the term frequency of q_i in the document D, $|D|$ is the number of words in the document D, $avgdl$ is the average number of words in the collection of documents. k_1 and b are parameters, by default $k_1 = 1.2$ and $b = 0.75$. $IDF(q_i)$ is the inverse document frequency weight of the query term q_i, calculated as:

$$IDF(q_i) = ln\left(\frac{N - n(q_i) + 0.5}{n(q_i) + 0.5} + 1\right)$$

where N is the number of documents in the collection and $n(q_i)$ is the number of documents containing the query term q_i

BM25F is derived from BM25 and it is used to apply the scoring when using multiple fields, which can possibly have different degrees of importance, term relevance saturation and length normalization. We can define BM25F score as a linear combination of BM25 scores on multiple fields.

Given $BM25_i$ score of the i^{th} field and a vector a of score boosters:

$$BM25F = \sum_i a_i \cdot BM25_i.$$

2.2 Rocchio Algorithm

The Rocchio Algorithm is an algorithm for implementing relevance feedback developed using the Vector Space Model as its basis.

The formula for the Rocchio Algorithm is:

$$\overrightarrow{Q_m} = \left(a \cdot \overrightarrow{Q_O}\right) + \left(b \cdot \frac{1}{|D_r|} \cdot \sum_{\overrightarrow{D_j} \in D_r} \overrightarrow{D_j}\right) - \left(c \cdot \frac{1}{|D_{nr}|} \cdot \sum_{\overrightarrow{D_k} \in D_{nr}} \overrightarrow{D_k}\right)$$

where $\overrightarrow{Q_m}$ is the modified query vector, $\overrightarrow{Q_O}$ is the original query vector, $\overrightarrow{D_i}$ is the document vector for the i^{th} document, D_r is the set of relevant documents, D_{nr} is the set of non-relevant documents and a, b and c are weight parameters.

2.3 Reciprocal Rank Fusion

Reciprocal Rank Fusion, or RRF, is a simple method for combining document rankings from multiple IR systems, and it usually yields better results than any individual system. RRF sorts the documents according to a naive scoring formula.

Given a set of documents D and a set of rankings R for the documents, the formula for RRF is:

$$RRFscore(d \in D) = \sum_{r \in R} \frac{1}{k + r(d)}$$

where k is a fixed number.

The formula is based on the idea that, while highly-ranked documents are more important, lower-ranked documents are still relevant. The constant k mitigates the effect of high ranking outliers.

2.4 Evaluation Measures

The following are measures used in the evaluation of Retrieval Effectiveness.

2.4.1 Precision at depth

Precision at depth, or P@d, the fraction of documents that are relevant among the first d documents retrieved. If r_i indicates the relevance of the i^{th} ranked document scored as 1 if relevant or 0 otherwise then:

$$P@d = \frac{\sum_{i=1}^{d} r_i}{d}.$$

2.4.2 Mean Average Precision

Average Precision, or AP, is calculated for a single query by taking the set of ranks at which the relevant documents occur, calculating the precision at those depths in the ranking, and then averaging the set of precision values obtained.

$$AP = \frac{1}{R} \sum_{i=1}^{d} \left(\frac{r_i}{i} \cdot \sum_{j=1}^{i} r_j \right)$$

Where r_i indicates the relevance of the i^{th} ranked document scored as 1 if relevant or 0 otherwise, d is the number of retrieved documents and R is the number of relevant documents for the query.

Mean Average Precision, or MAP, is the mean of AP scores for a query batch, so where Q is the number of queries:

$$MAP = \frac{\sum_{i=1}^{Q} AP_i}{Q}.$$

2.4.3 Binary preference

Binary Preference, or BPref, only uses information from judged documents.

It is a function that measures how frequently relevant documents are retrieved before non-relevant ones.

$$BPref = \frac{1}{R} \sum_{i=1}^{d'} \left[r'_i \cdot \left(1 - \frac{i - \sum_{j=1}^{i} r'_j}{N} \right) \right]$$

where, considering the vector of relevance values removing the unjudged documents, d' is the number of judged documents retrieved, r'_i is the relevance of the i^{th} ranked judged document and N is the number of documents judged as not-relevant.

When the judgements are complete BPref is equivalent to MAP, with incomplete judgements BPref is shown to be more stable.

Chapter 3

Data

The search and evaluation are performed using the datasets of the fifth and final challenge round.

The datasets from the fourth challenge round are also used to choose the ideal field boosting and to build the engine using relevance feedback.

Four types of datasets are used for this study:

CORD-19 - collection of documents to perform the information retrieval on;

List of valid doc-ids - list of IDs that identify which documents in the CORD-19 collection are to be used in a round;

Topic set - list of topics that determine what query we are inputting in the IR system in a round;

Relevance Judgements - list of IDs paired with relevance judgements that identify whether a document is relevant to a topic.

All of the above are publicly available, all CORD-19 releases can be downloaded from the CORD-19 database mantained by AI2, while all the other datasets for each round (together with what release of CORD-19 has been used for each round) can be downloaded from NIST's TREC-COVID database.

3.1 Data Structure

The **CORD-19 document collection** is available as a .csv file.

It contains a list of documents with the following fields: cord_uid, sha, source_x, title, doi, pmcid, pubmed_id, license, abstract, publish_time, authors, journal, mag_id,
who_covidence_id, arxiv_id, pdf_json_files, pmc_json_files, url, s2_id.

The fields that have been used are:

Cord_uid an alphanumeric string that defines an unique ID for each document in the collection;
Title of the document;
Abstract of the document;
Publish_time that defines the date of publication of the document.

The **list of valid doc-ids** is a .txt file that contains the list of cord_uid to identify which documents to include in the run.

The **topic set** is a .xml file that contains a set of topics (30 in the first round, then 5 more added for each subsequent round, up to 50 for the fifth round), each topic has 3 fields:

Query provides short statement containing only the keywords regarding information needed;
Question provides a more complete description of the information needed;
Narrative provides extra clarification, mainly used to understand what information the judges should consider relevant to the topic.

The **relevance judgements** are available as a .txt tile. Each row is a document that has been judged. There are four columns: the first is the number of topic the judgement is performed on, the second identifies at which point in time the judgement has been done, the third corresponds to the cord_uid of the document being judged. Finally, the last one is a thricotomous variable for the relevance judgement: 2 for relevant, 1 for partially relevant (which, however, has the same value of 2 for scoring runs), 0 for not relevant.

3.2 Data Transformation and Cleaning

In order to use the CORD-19 collection and the topic set with more ease, they have been transformed to .json files.

The collection has been transformed using a python program [6.1], while the topic set has been transformed using a free online tool.

The CORD-19 collection is also missing data, which are fundamental for the correct function of the information retrieval when using the publish_time field.

For some documents such field is blank. In this case, the chosen solution was to manually input as publish_time, for those few documents, the release date of the CORD19 dataset used.

If a considerable amount of documents were to miss such field, it would be advisable to write a program to perform the cleaning.

Chapter 4

Experiment's structure

The search engine of choice for the experiments is Elasticsearch.

Elasticsearch is a free and open source full-text search engine based on the Lucene library, with a HTTP web interface and schema-free JSON documents.

4.1 Database Creation and Indexing

Except for the base run, which uses Elasticsearch's standard analyzer for *text* fields, custom analyzers are applied to the database, which needs to be done by mapping.

Mapping allows to define the field type an the analyzer uses for a certain field. Moreover, it needs to be done before indexing, as the filter is used during indexing to determine what value to insert in each field.

The following token filters are used:

Synonym_graph custom filter that allows to define synonyms. It has been used to transform all the other words to the first one, the list of synonymous terms applied is "[coronavirus, covid 19, sars cov 2, 2019 ncov]". The purpose of this filter is to manually solve the biggest vocabulary mismatch problem present in the test collection, which influences the searches on all queries;

Lowercase transforms all upper case characters to lower case;

Stop is the standard filter to remove English stopwords;

Stemmer provides algorithmic stemming, the standard for English is porter stemming algorithm;

Shingle used when performing the search on bigram instead of tokens. It produces word n-grams by concatenating adjacent tokens. It has been cus-

tomized to only output two-word shingles, since the filter would also output unigrams by default.

The filters are applied to the "title" and "abstract" fields.

An example of the database creation with mapping is available at [6.2]. In windows environment the ' characters would need to changed to " and the " inside them to \".

The document collection has to be indexed in the created database, which is done with a python program [6.3]. First of all, before inserting a document in the database, a check is performed to establish whether the document's value for the cord_uid field is in the list of valid doc-ids.

In the event that it is, the document is added to the database; the relative cord_uid is removed from the list in order to prevent cases of duplicated doc_uids (which would break the evaluation process as only one ID should be found for a topic in a run).

Alternatively, the document is skipped.

4.2 Search and Output Structure

The next step consists of searching the documents and outputting the ones that are deemed most relevant by the search engine.

The search and output for all the topics is done by using a python program [6.4]. The program has three global variables:

Indexname name of the elasticsearch index to perform the search on;

Size maximum number of documents retrieved for each topic search, 1000 is the maximum allowed (and suggested amount) for this challenge;

Origin date used in the *distance feature* query to calculate distances. The date of CORD-19 release is chosen for the searches; in a real world application the current date would be used instead;

Querytype allows to choose between *query*, *question* or *narrative* field from the topic set for the search.

The search is executed using a should query, which allows a document to be retrieved even if its tokens only match one token of the search query.

Every document is scored using BM25 (BM25F in case of searching in multiple fields) and, then, the documents are retrieved in descending score order.

As can be seen in the code example [6.4], the engine allows to: select different boosts for the fields being matched (giving a field more weight than the others), add a different boost to each document based on the date of publication, filter the search in order to only consider documents presenting a certain token in a field and filter the search to only consider documents published in a certain date range.

Lastly, the program outputs the retrieved documents to a text file in TREC's format, where every line is in the form:

Topicid is the number of the topic;
Q0 is the literal "Q0", unused column for this challenge.
Docid is the cord_uid of the document retrieved;
Rank is the rank position of the document in the list;
Score is the similarity score computed by the system;
Run-tag is a name assigned to the run.

4.3 Relevance Feedback

Relevance Feedback in the experiments has been built using the Rocchio Algorithm with (0,1,0) as vector of parameters. Therefore, only tokens from relevant documents are used as query; neither the original query nor tokens from non-relevant documents are used.

The search using relevance feedback uses a different python program [6.5].

An ulterior global variable is present: **Indextrain**. It identifies the database used to train the algorithm.

The program also receives in input the relevance judments relative to the version used in the training database.

For each topic, the program searches the *termvectors* of relevant documents and saves the tokens and respective cumulative weight in the documents, up to a maximum of 1000 tokens (the maximum number needed to be put in place due to Elasticsearch's limitations).

Furthermore, since topic sets are super-sets of the previous ones, for each topic in the training database, the search is performed using the tokens collected instead of the queries offered by the topic set. The score obtained by each token is boosted by the respective cumulative weight.

Topics not in the training topic set are instead searched like in the previous program. The output is done identically to the previous.

Only *termvectors* deriving from the title field have been used. Moreover, only documents judged as fully relevant have been considered. Due to Elasticsearch's limitations, neither partially relevant documents nor non-relevant ones, as negative boosting, have not been used.

4.4 Evaluation Structure

Each run's evaluation is performed using trec_eval.

Trec_eval is the standard tool used by the TREC community for evaluating an ad hoc retrieval run. It uses the output file and the set of relevance judgements to return, in output, the results of the run as a list of standard relevant measures.

The measures considered for the improvement of effectiveness in this study are: *P@20*, *MAP* and *BPref*.

Chapter 5

Experimental evaluation

5.1 Baseline

The baseline run uses Elasticsearch's standard analyzer. Because of it, there is no specific need to manually map any field of the database.

The search is performed on both the *title* and the *abstracts* with no boosting applied. Therefore, the fields hold the same weight in document relevance scoring.

The search is performed thrice, once for each of the different topic set available field.

As expected, using the *narrative* fields, compared to the others, returns much worse results and it is, in fact, not supposed to be used for this purpose. Hence, no runs using the *narrative* field will be included.

The baseline runs using *query* and *question* fields show comparable results. However, only the *question* field will be considered as the baseline run, since its measures of evaluation are slightly better.

Searches using the *query* field will be considered in the following runs, instead.

The following are the evaluation measures for the baseline run:

P@20	MAP	BPref
0.2360	0.0695	0.3106

5.2 Customized Analyzer and Filters

Custom Analyzer

The first step to improve the IR is applying a customized analyzer.

Before indexing, the database is mapped in order to use the *synonym_graph*, *lowercase*, *stop* and *stemmer* token filters, for the *title* and *abstract* text fields.

The search then continues like the baseline.

The following are the evaluation measures for the runs with customized analyzers using the *query* and *question* fields respectively:

Run	P@20	MAP	BPref
Query	**0.2840**	**0.1037**	**0.3482**
Question	0.2380	0.0879	0.3416

Filter

Subsequently, filters are applied to the search engine.

The filters applied are a term filter and a date filter. The former filters only documents that present the word "coronavirus" in the *title* or the *abstract* field. The latter filters only documents published after 01/11/2019.

Both filters used singularly improve the search's effectiveness, with the date filter being more effective. However, using both filters is less effective than filtering only by date.

The following are the evaluation measures, compared to the runs without filters:

Run	P@20	MAP	BPref
Query	0.2840	0.1037	0.3482
Query + term	0.3120	0.1182	0.3718
Query + date	**0.3130**	**0.1231**	0.3823
Query + term & date	0.3120	0.1227	0.3769
Question	0.2380	0.0879	0.3416
Question + date	0.2740	0.1099	**0.3896**

5.3 Field Boosting

Another way to further improve the engine would be by establishing the ideal boosting parameters for the *title* and *abstract* field.

However, using the same database for both the search and the choice of the parameters would lead to overfitting. In order to avoid that, the parameter choice is done on the previous step of the TREC-COVID challenge.

On the fourth TREC-COVID challenge step, the same procedures of the filtered search are applied for a multitude of runs, while also applying different boostings on the *title* or *abstract* fields.

A total of 82 runs are performed: 41 using the *query* and 41 using the *question* field. For each of them, a run is performed without boosting, 20 runs boosting the *title* field by 0, 0.05, 0.10, ..., 0.95 and 20 boosting the *abstract* field by 0, 0.05, 0.10, ..., 0.95.

The runs returning the best results are the ones with a 0.4 boost on the *title* field regarding *query* and 0.75 boost on the *title* field regarding *question*.

The following are the evaluation measures, comparing the previous best runs (with date filters) and their respective runs when the ideal boostings found are applied:

Run	P@20	MAP	BPref
Query + date	0.3130	0.1231	0.3823
Query + date w/ 0.4 title	**0.3160**	**0.1284**	0.3811
Question + date	0.2740	0.1099	**0.3896**
Question + date w/ 0.75 title	0.2800	0.1119	0.3878

5.4 Bigrams

Instead of the standard bag-of-words format, which uses single words tokens, the next method uses bigrams, which are sequences of two adjacent tokens).

In order to do so, the *title* and *abstract* fields are mapped with an ulterior filter called *shingle*, customized to output only bigrams and not unigrams.

The documents have to be indexed again, and the other procedures applied are the same of the previous step.

Only the *question* field is used in the bigrams run, since the query field is written in a non-natural language format.

The evaluation measures for the bigrams run are:

P@20	MAP	BPref
0.2160	0.0484	0.1759

The run has a much lower amount of relevant retrieved documents and, overall, lower evaluation measures. It is, however, comparable regarding evaluation measures when only taking in account the first documents found. Hence, showing that the engine using bigrams is still effective for the first retrieved documents.

Although the bigram run is not effective by itself, it still might be useful when applying RRF.

5.5 Relevance Feedback

A completely different technique is using Relevance Feedback.

Relevance Feedback utilizes information collected previously to improve the search performing query expansion.

In this case the relevance judgements from the previous rounds are used to select relevant documents from the collection. The tokens contained in the title of the relevant documents are collected and used as queries, with boosts relative to the tokens' cumulative weight.

Since previous rounds' topic sets are a subsection of the latest one, Relevance Feedback cannot be applied to the last 5 topics. Therefore, for those topics, the search will be the same used earlier.

The following are the evaluation measures for the run using RF:

P@20	MAP	BPref
0.3030	0.1536	0.5149

Relevance Feedback brings to a huge improvement in the search engine effectiveness.

5.6 Distance Feature

An additional way to improve the engine, particularly when considering document collections with a rapid expansion, might be including a boost based on how recent the document is.

Distance_feature allows to give more weight to documents closer to a certain date or location.

The *origin*, which is the date that allows to reach the maximum boosting, is selected as the date of the CORD-19 database release. In a real-world scenario, on the other hand, it would use the current date.

A parameter called "*pivot*" is required. It marks the distance in which the boost is halved compared to the boost at the *origin*. In this case the pivot has been selected as a standard of 60 days.

The following are the evaluation measures for the precedent runs and a comparison when adding the distance feature boosting:

Run	P@20	MAP	BPref
Query + date w/ 0.4 title	0.3160	0.1284	0.3811
Query + date w/ 0.4 title & distance	**0.3240**	0.1309	0.3833
Question + date w/ 0.75 title	0.2800	0.1119	0.3878
Question + date w/ 0.75 title & distance	0.2760	0.1131	0.3894
RF	0.3030	**0.1536**	0.5149
RF w/ distance	0.3020	0.1532	**0.5152**

The results vary and no overall improvement can be found using the distance feature.

5.7 RRF

RRF allows to combine the results of different runs to improve the effectiveness. The following are the measures for various applications of RRF:

Runs combined	P@20	MAP	BPref
Query + date, Question + date	0.3010	0.1320	0.4115
Query + date w/ 0.4 title, Question + date w/ 0.75 title	0.3140	0.1369	0.4128
Query + date, Question + date, Query + date w/ 0.4 title, Question + date w/ 0.75 title	0.3160	0.1380	0.4143
Query + date w/ 0.4 title, Question + date w/ 0.75 title, RF	0.3210	0.1661	0.5038
Query + date, Question + date, Query + date w/ 0.4 title, Question + date w/ 0.75 title, RF, RF w/ distance	0.3230	**0.1671**	**0.5049**
Query + date, Question + date, Query + date w/ 0.4 title, Question + date w/ 0.75 title, RF, RF w/ distance, bigram	**0.3270**	0.1652	0.4989

Chapter 6

Conclusions

Below, is shown the baseline evaluation measures, compared to the evaluation measures from the three best measuring search engines, which combine in them all the techniques applied:

Run	P@20	MAP	BPref
Baseline	0.2360	0.0695	0.3106
RF w/ distance	0.3020	0.1532	**0.5152**
Query + date, Question + date, Query + date w/ 0.4 title, Question + date w/ 0.75 title, RF, RF w/ distance	0.3230	**0.1671**	0.5049
Query + date, Question + date, Query + date w/ 0.4 title, Question + date w/ 0.75 title, RF, RF w/ distance, bigram	**0.3270**	0.1652	0.4989

We can notice an improvement of 38.6%, 140.4% and 65.9% respectively for the three measures.

The techniques that offered the best improvement are Relevance Feedback, Reciprocal Rank Fusion, custom analyzers and filtering; whereas field boosting, bigrams and distance feature can be considered negligible.

When comparing the results to the best official runs we can notice that the *BPref* found here is comparable (90% of BPref $\in$ [0.3925,0.6091]), while *P@20* and *MAP* are far inferior (90% of P@20 $\in$ [0.665,0.846] and 90% of MAP $\in$ [0.22,0.4169]).

The reason for this is that, as mentioned before, P@20 and MAP consider unjudged documents as non-relevant. In fact, the documents have been judged based on the top documents belonging to the official runs. Hence, it is be expected that more top documents for those runs are judged, compared to the top

documents in this study's runs. BPref is comparable as it only considers judged documents, instead.

While there definitely are decent results, the lack of machine and deep learning techniques is noticeable. This is particularly evident when comparing to the very top runs' measurement, which make use of such techniques.

These techniques would definitely be topics to expand on in future works.

Code

```
import csv
import json


csvfile = open('metadata_rnd4.csv', 'r', encoding="UTF-8")
jsonfile = open('metadata_rnd4.json', 'w')


reader = csv.DictReader(csvfile)
for row in reader:
    json.dump(row, jsonfile)
    jsonfile.write('\n')
```

Listing 6.1: Python code to trasform .csv to .json.

```
curl -XPUT "http://localhost:9200/covid_rnd5_an?pretty" -H "
   Content-Type: application/json" -d'{
  "settings": {
    "index": {
      "analysis": {
        "filter": {
          "graph_syn": {
            "type":"synonym_graph",
            "expand":"false",
            "synonyms": ["coronavirus, covid 19, sars cov 2, 2019
    ncov"]
          },
          "my_shingle": {
            "type":"shingle",
            "output_unigrams":"false"
          }
        },
        "analyzer": {
          "cord_analyzer": {
            "type": "custom",
```

```
            "tokenizer": "standard",
            "filter": ["lowercase","stop","stemmer","graph_syn","
    my_shingle"]
          }
        }
      }
    }
  },
  "mappings": {
    "properties": {
      "title": {
        "type": "text",
        "analyzer": "cord_analyzer"
      },
      "abstract": {
        "type": "text",
        "analyzer": "cord_analyzer"
      }
    }
  }
}'
```

Listing 6.2: Example of database creation with analyzers.

```
#--- importazione di moduli interi
import sys
import json

#--- importazione di parti di modulo ---#
from elasticsearch import Elasticsearch

#--- variabili globali ---#
INDEXNAME = 'covid_rnd5_bigram'
DOCTYPE   = '_doc'
VALIDNAME = 'docids-rnd5.txt'

#--- connettiti al server
es = Elasticsearch([{'host':'localhost','port':9200}])

indexes=[]

with open(VALIDNAME) as f:
    indexes = f.read().splitlines()

#--- scansione dei file dei documenti ---#
```

```
nargs = len(sys.argv)
ndoc = 0
for i in range(1,nargs):
    #--- apertura del file del documento ---#
    print("processing",sys.argv[i],"...")
    infile = open(sys.argv[i],'r', encoding="utf8")
    #--- lettura del documento ---#
    for doc in infile:
        if len(doc) > 0:
            ndoc += 1
            record = json.loads(doc)
            print(sys.argv[i],ndoc,'...',end='')
            if record["cord_uid"] in indexes:
                indexes.remove(record["cord_uid"])
                res = es.index(index=INDEXNAME,
                               doc_type=DOCTYPE,
                               id=ndoc,
                               body=record)
                print("added")
            else:
                print("id not valid")
    print("done")
print("done")
```

Listing 6.3: Example of python code to index the database.

```
import sys
import json
import numpy as np

from elasticsearch import Elasticsearch as server

#--- questa funzione sara' usata piu' avanti ---#
def res(results, query = "1", n = 10, tag = "jackdiquadri"):
    rank = 0
    for hit in response['hits']['hits']:
        rank += 1
        print(query,"Q0",hit["_source"]['cord_uid'],rank,hit['
    _score'],tag,sep='\t')

#--- variabili globali ---#
INDEXNAME = 'covid_rnd5_bigram'
SIZE    = 1000
ORIGIN    = "2020-07-16"
QUERYTYPE = 'query'
```

```
#ORIGIN    = "2020-06-19"

client = server(['localhost:9200'])

infile = open(sys.argv[1],'r')
queries = json.loads(infile.read())
queries_list = queries['topic']
for query in queries_list:
    num = query['@number']
    text = query[QUERYTYPE]
    query_dict = {
        "query": {
            "bool": {
                "should": [
                    { "match": { "title"  : { "query" : text, "boost" :0.4} } },
                    { "match": { "abstract" : { "query" : text, "boost" :1} } },
                    {"distance_feature":{"field":"publish_time","pivot":"60d","origin":ORIGIN}}
                ],
                "filter":[
#                    {"term": { "title" : "coronaviru"} or { "abstract" : "coronaviru"}},
                    {"range": {"publish_time": {"gte": "2019-11-01"}}}
                ]
                }
            }
        }

    response = client.search(index=INDEXNAME,body=query_dict,size=SIZE)
    res(response['hits']['hits'],num,SIZE)
```

Listing 6.4: Example of python code for search and output of relevant documents.

```
import sys
import json
import csv
import numpy as np
from elasticsearch import Elasticsearch as server

infile = open(sys.argv[1],'r',encoding="utf-8")
qrels  = list(csv.reader(infile, delimiter=" "))
```

```
9
10 INDEXNAME = 'covid_rnd5_an'
11 INDEXTRAIN= 'covid_rnd4_an'
12 SIZE    = 1000
13 ORIGIN    = "2020-07-16"
14 #ORIGIN    = "2020-06-19"
15 nq1       = 45
16 nq2       = 50
17 rel       = [[],[],[]]
18 all_terms = [[],[],[]]
19 lista     = []
20 client = server(['localhost:9200'])
21
22 for i in range(nq1):
23     rel[0].append([])
24     rel[1].append([])
25     rel[2].append([])
26     all_terms[0].append({})
27     all_terms[1].append({})
28     all_terms[2].append({})
29
30 for i in range(nq2):
31     lista.append([])
32
33 for j in qrels:
34     rel[int(j[3])][int(j[0])-1].append(j[2])
35
36
37 #for i in range(len(rel)):
38 for i in range(2,len(rel)):
39     for j in range(len(rel[i])):
40         for k in rel[i][j]:
41             finduid={"query": {"term": {"cord_uid": {"value": k
   }}}}
42             trainresponse = client.search(index=INDEXTRAIN,body=
   finduid,size=1)
43             if len(trainresponse['hits']['hits'])>0:
44                 docid=trainresponse['hits']['hits'][0]['_id']
45                 tv=client.termvectors(index=INDEXTRAIN, id=docid,
    fields=['title'], term_statistics=True)
46                 for field in tv['term_vectors']:
47                     terms = tv['term_vectors'][field]
48                     N = terms['field_statistics']['doc_count']
49                     occurrences = terms['terms']
```

```
50                      for term in occurrences:
51                          if len(all_terms[i][j])<1000:
52                              trm = occurrences[term]
53                              if term in all_terms[i][j]:
54                                  all_terms[i][j][term]=all_terms[i
    ][j][term]+float(trm['term_freq'])
55                              else:
56                                  all_terms[i][j][term]=float(trm['
    term_freq'])
57
58 def res(results, query = "1", n = 10, tag = "jackdiquadri"):
59     rank = 0
60     for hit in results:
61         rank += 1
62         print(query,"Q0",hit["_source"]['cord_uid'],rank,hit['
    _score'],tag,sep='\t')
63
64 for j in range(2,len(all_terms)):
65     for i in range(nq1):
66 #         print(" ".join(all_terms[j][i]).encode("utf-8"))
67         for key in all_terms[j][i]:
68             lista[i].append({ "multi_match": {"fields":["title","
    abstract"],"query":key,"boost":all_terms[j][i][key]}})
69         lista[i].append({"distance_feature":{"field":"
    publish_time","pivot":"60d","origin":ORIGIN}})
70 #         lista[i].append({ "multi_match": {"fields":["title
    ^0.75","abstract"],"query":" ".join(all_terms[j][i])}})
71
72
73 infile  = open(sys.argv[2],'r')
74 queries = json.loads(infile.read())
75 queries_list = queries['topic']
76 for i in range(nq1,nq2):
77     text = queries_list[i]['query']
78     lista[i].append({ "multi_match": {"fields":["title","abstract
    "],"query":text}})
79     lista[i].append({"distance_feature":{"field":"publish_time","
    pivot":"60d","origin":ORIGIN}})
80
81
82 for i in range(nq2):
83     q = {
84         "query":{
85             "bool":{
```

```
                "should":lista[i],
                "filter":[
#                    {"term": { "title" : "coronaviru"} or { "abstract" : "coronaviru"}},
                    {"range": {"publish_time": {"gte": "2019-11-01"}}}]}}}


    response = client.search(index=INDEXNAME,body=q,size=SIZE,request_timeout=10)
    res(response['hits']['hits'],i+1,SIZE)
```

Listing 6.5: Example of python code for search and output of relevant documents using relevance feedback.

```
import sys
import json
import csv

nq=50
d=[]
nargs = len(sys.argv)
for i in range(1,nargs):
    with open(sys.argv[i],'r', encoding="utf8") as infile:
        reader=csv.reader(infile, delimiter="\t")
        d.append(list(reader))

diz=[]
rrfdocs=[]

for i in range(nq):
    diz.append({})
    rrfdocs.append([])


def append_value(dict_obj, key, value):
    if key in dict_obj:
        dict_obj[key].append(value)
    else:
        dict_obj[key] = [value]

for z in range(len(d)):
    for i in range(len(d[z])):
        append_value(diz[int(d[z][i][0])-1],d[z][i][2],[z,i])

```

```
for z in range(nq):
    k=0
    for i in diz[z]:
        rrfdocs[z].append([i])
        for j in diz[z][i]:
            rrfdocs[z][k].append(1/(30+int(d[j[0]][j[1]][3])))
        rrfdocs[z][k][1] = sum(rrfdocs[z][k][1:])
        k=k+1

    rrfdocs[z].sort(key=lambda x: x[1], reverse=True)


for z in range(nq):
    for i in range(min(len(rrfdocs[z]),1000)):
        print(z+1,"Q0",rrfdocs[z][i][0],str(i+1),rrfdocs[z][i
   ][1],"jackdiquadri",sep='\t')
```

Listing 6.6: Python code to apply Reciprocal Rank Fusion.

Bibliography

[1] Chen J. S., Hersh W. R., *A comparative analysis of system features used in the TREC-COVID information retrieval challenge*, journal of Biomedical Informatics Vol.117, `https://www.sciencedirect.com/science/article/pii/S1532046421000745?via%3Dihub`, 2021

[2] Melucci M., *INFORMATION RETRIEVAL. Metodi e modelli per i motori di ricerca*, FrancoAngeli, 2013.

[3] Moffat A., Zobel J., *Rank-Biased Precision for Measurement of Retrieval Effectiveness*, ACM Transactions on Information Systems Vol.27, `https://people.eng.unimelb.edu.au/jzobel/fulltext/acmtois08`, 2008.

[4] *Elastisearch Documentation*, `https://www.elastic.co/guide/index.html` (Last accessed: 08/2021).

[5] *TREC-COVID Home*, `https://ir.nist.gov/covidSubmit/index.html` (Last accessed: 02/2022)